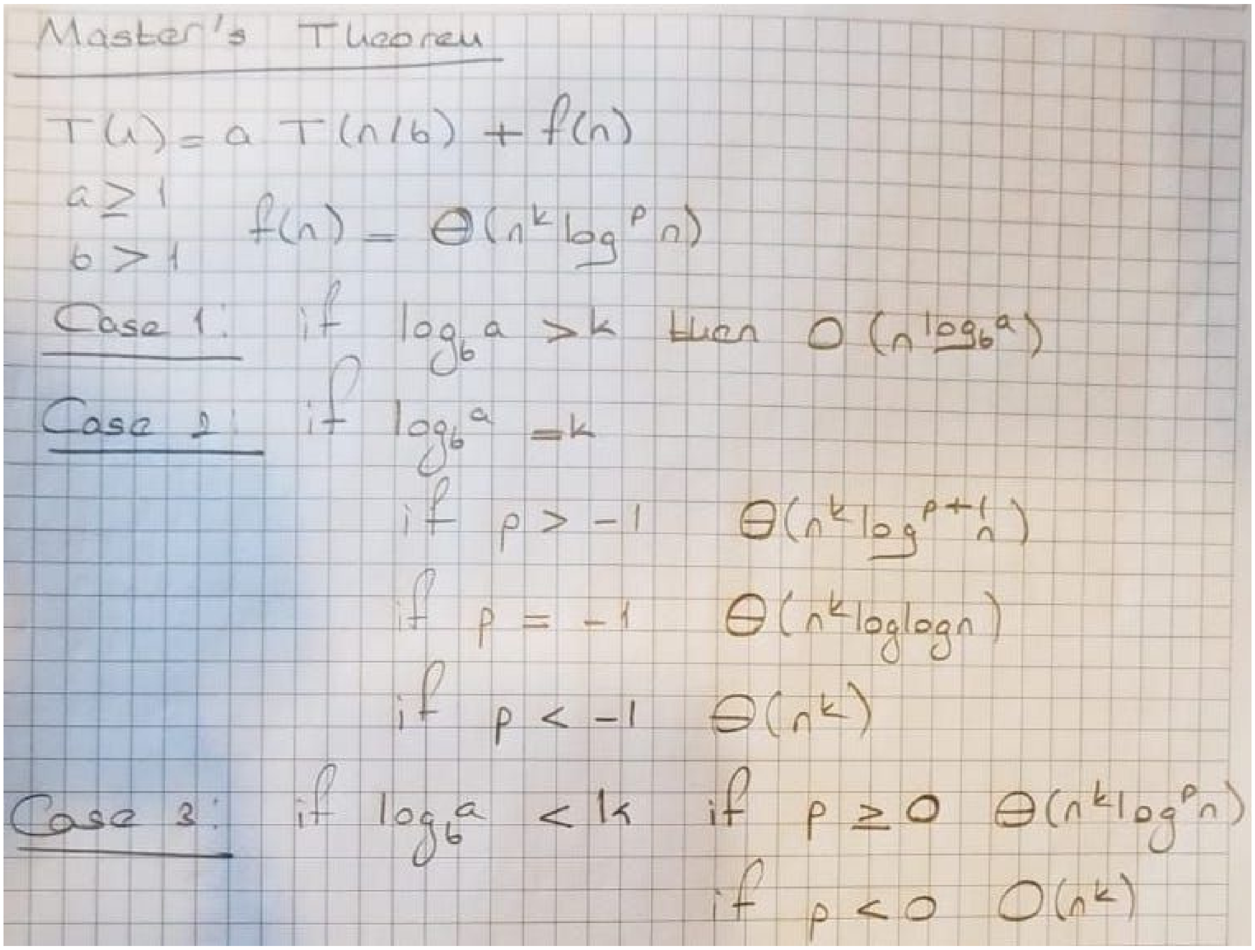
Semih Balki - 19010

P1)

2^(2^n) > n! > n^100 > n.(2^n) > 2^n > (log(n))! > n.log(n) > log(n!) > n > log(n)

P2)



Master's Theorem

$T(n) = a\,T(n/b) + f(n)$

$a \geq 1$

$b > 1$

$f(n) = \Theta(n^k \log^p n)$

Case 1: if $\log_b a > k$ then $O(n^{\log_b a})$

Case 2: if $\log_b a = k$

if $p > -1$ $\Theta(n^k \log^{p+1} n)$

if $p = -1$ $\Theta(n^k \log\log n)$

if $p < -1$ $\Theta(n^k)$

Case 3: if $\log_b a < k$ if $p \geq 0$ $\Theta(n^k \log^p n)$

if $p < 0$ $O(n^k)$

Figure 1: Above image is the Master's Theorem, put as an image since to show the notations.

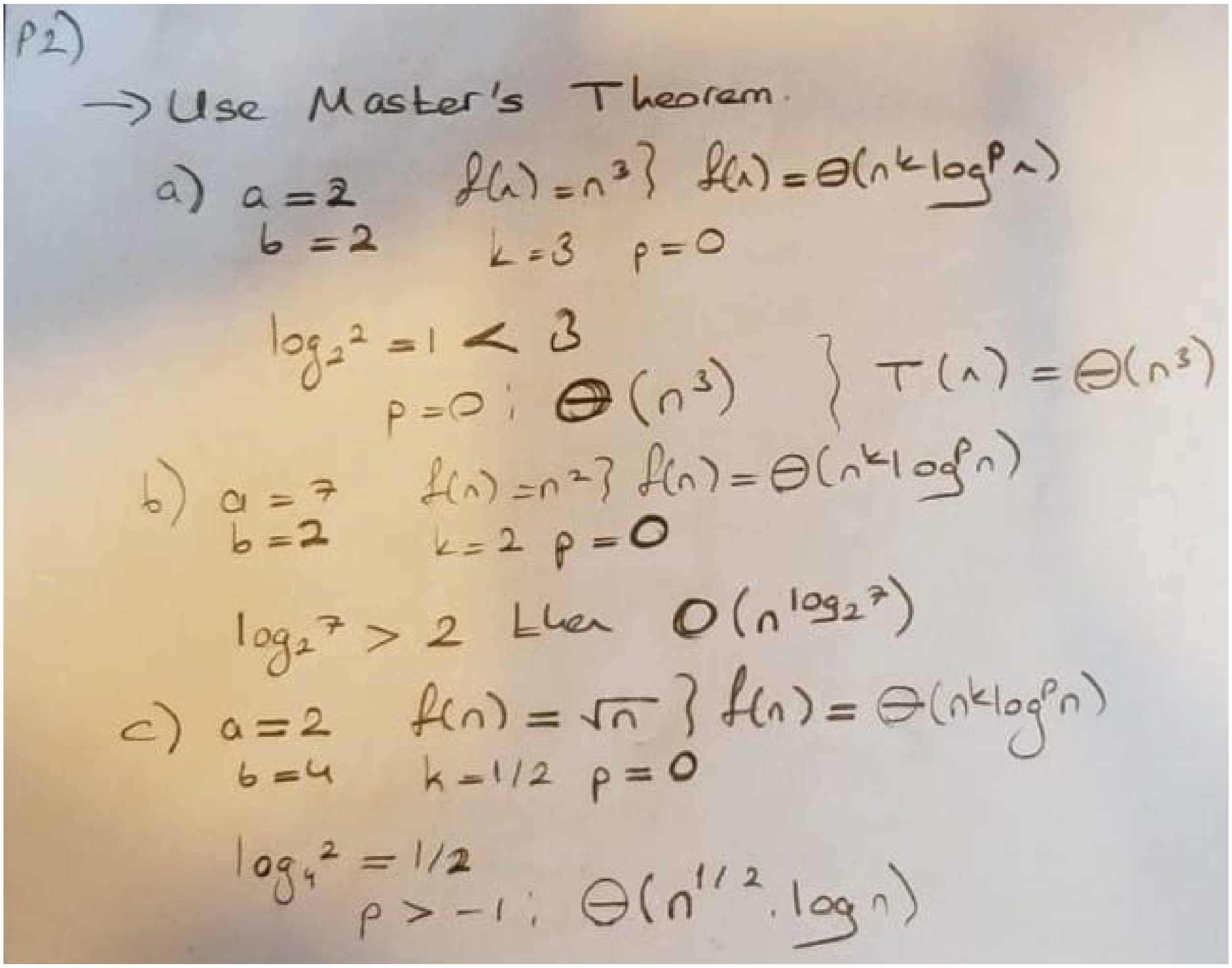


Figure 2: The (a), (b) and (c) of the Problem 2.

As we can observe that at (a), (b) and (c) of the Problem 2 are can be solved by Master's Theorem.

d)

For d, we can not use Master's Theorem; use proof by substitution method:

Guess: T(n) = O(n)

Assume: T(k) <= c.k for k < n and try to prove T(n) <= c.n

T(n) = T(n - 1) + n

<= (c.n - 1) + n

= cn - (c - n)

c.n is the desired part

(c - n) is the residual part

c - n >= 0 if c >= n

We proved that T(n) = O(n)

P3)

a)

i)

Assume that the size of the array is a power of 2, say 2^k. Each time around the loop, when we examine the middle element, we cut the size of the subarrays we look at in half. So before the first iteration the size of the subarray if interest is 2^k.

After the second iteration it is of size 2^(k-1), then 2(k-2) etc.

After k iterations it will be 2^(k-k)=1, so we stop after next iteration.

Thus, overall when given a size of array n we perform log(n) + 1 operations.(+1 comes from calculating the length of the array(len(alist))).Then it is O(logn)

ii)

ii)

$$T(n) = \begin{cases} 1, & n = 1 \\ T(n/2) + n + 1, & n > 1 \end{cases}$$

Actually equation is $T(n) + O(n) + 1$

interpreted $O(n)$ as n at the equation.

→ To solve recurrence, use substitution method:

$$T(n) = T(n/2) + n + 1 \qquad T(n/2) = T(n/4) + n/2 + 1$$
$$= T(n/4) + 3n/2 + 2 \qquad T(n/4) = T(n/8) + n/4 + 1$$
$$= T(n/8) + 7n/4 + 3$$
$$\vdots$$
$$= T(n/2^k) + \frac{(2^k - 1)\cdot n}{2^{k-1}} + k$$

$$\Rightarrow T(\underbrace{n/2^k}_{1}) = 1 \Rightarrow \frac{n}{2^k} = 1 \Rightarrow n = 2^k$$
$$k = \log n$$

→ Plug in $k = \log n$:

$$T(n) = 1 + \frac{(2^{\log n} - 1)\cdot n}{2^{\log n - 1}} + \log n$$
$$= 1 + \frac{(n-1)\cdot \cancel{n}}{(\cancel{n}/2)} + \log n$$
$$= 1 + 2n - 2 + \log n$$
$$= 2n + \log n - 1$$
$$= O(n + \log n)$$

Figure 3: solving question by substitution method.

b)

i)

Algorithm	**N = 10^4**	**N = 10^5**	**N = 10^6**	**N = 10^7**
Iterative	**1.7982999999999194e-05**	**4.6781999999985644e-05**	**8.725899999983966e-05**	**0.000145233999999988307**
Recursive	**9.421399999998137e-05**	**0.00103663499999999525**	**0.01240415600000011**	**0.16323363600000018**

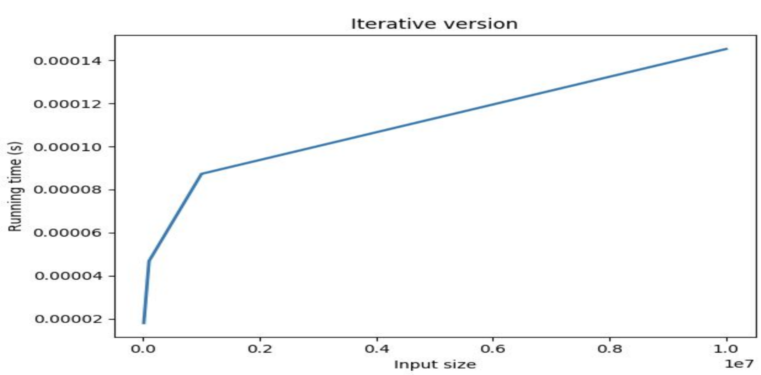
CPU: 2.9 GHz Intel Core i7

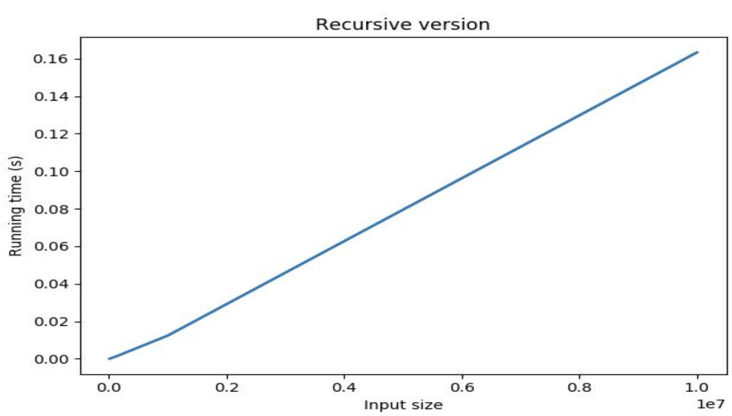
RAM: 16GB

OS: Mac OS

Python IDE: Pycharm

ii)





iii)

The experimental results of the given algorithms look like O(n) therefore in case of scalability given algorithms are efficient for huge numbers especially iterative version looks have smaller running time(s) thus iterative version looks more efficient.

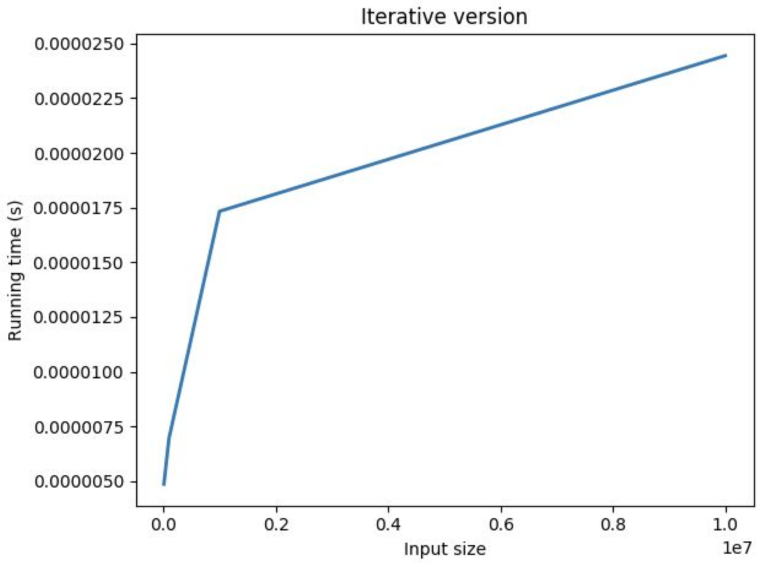
iv)

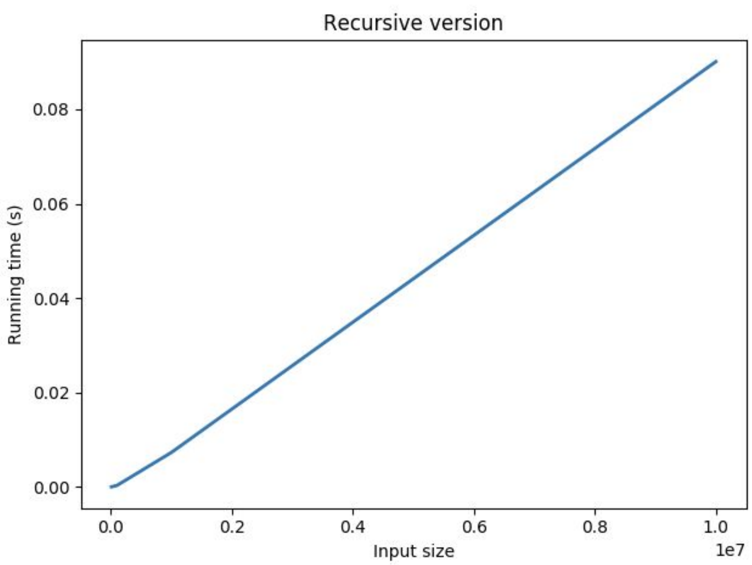
No, experimental results do not confirm with the theoretical results since theoretical results show complexity is O(logn), however, experimental looks like O(n).(But we cannot directly say this, to confirm this we need to do many more experiments)

c)

Algorithm	**N = 10^4**		**N = 10^5**		**N = 10^6**		**N = 10^7**	
	mean	**Std dev**	**mean**	**Std dev**	**mean**	**Std dev**	**mean**	**Std dev**
Iterative	**1.26307600000034157e-05**	**9.083256598563244e-05**	**7.036699999973805e-06**	**9.284459409459166e-05**	**2.456946000002347e-05**	**0.0001484611522516322**	**2.400319999998679e-05**	**0.00021429010453659207**
Recursive	**3.13445399999995643e-05**	**0.0007343193059683236**	**0.0003212335399999855**	**0.006224687140177192**	**0.00730638817999992**	**0.0891489289325592**	**0.0900918006399996**	**1.0621669376451024**

ii)





iii)

In terms of graphical recursive version do not have significant change(actually not at all) just the values are changed, however, at the iterative version exponential rate has increased at the beginning therefore we can say a change occurred at the iterative version.

d)

Since the cost of extracting a slice of a list using : (e.g., alist[0:n/2]) is O(n), we can do some trick to avoid such as; using the indexes of the array at each call.

```python
def binarySearch_improved(elem, arr, start, end):
    if start > end:
        return False
    mid = (start + end)//2
    if arr[mid] == elem:
        return mid
    elif arr[mid] > elem:
        # recurse to the left of mid
        return binarySearch_improved(elem, arr, start, mid - 1)
    else:
        # recurse to the right of mid
        return binarySearch_improved(elem, arr, mid + 1, end)
```

As you can see from the above image instead of using slicing function, we used start and end indexes of the array as parameter, that's how we avoid the cost of O(n).

Our aim is to improve the recursive so that it has the same asymptotic running time as the iterative one, and we can observe this experimentally.
Actually, at the first iteration the improved recursive one passed iterative one but former iterations passes the recursive one.

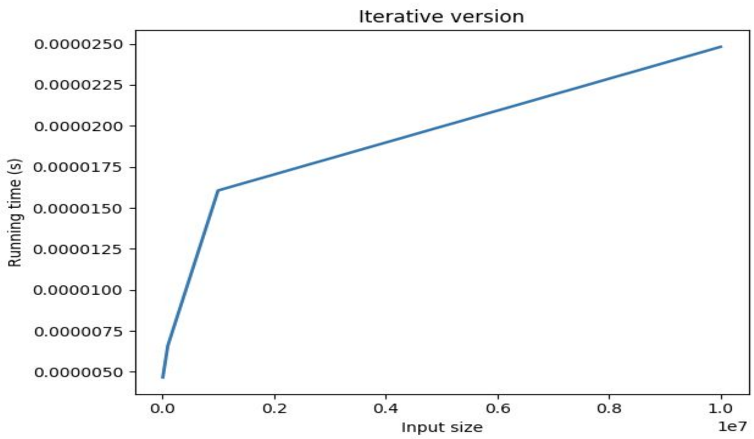
```
End of iteration:  1
True
End of iteration:  2
False
End of iteration:  3
False
End of iteration:  4
False
```

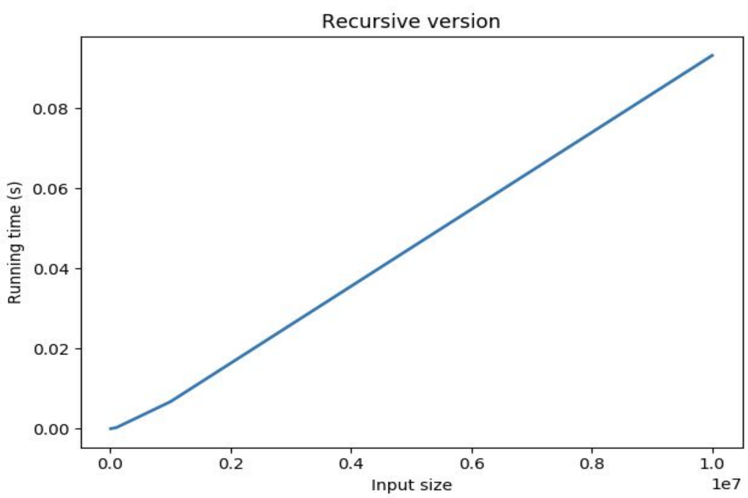
Note: Below if else statement is an explanation for 'True-False' at the above image
If running time of iterative < running time of improved recursive:

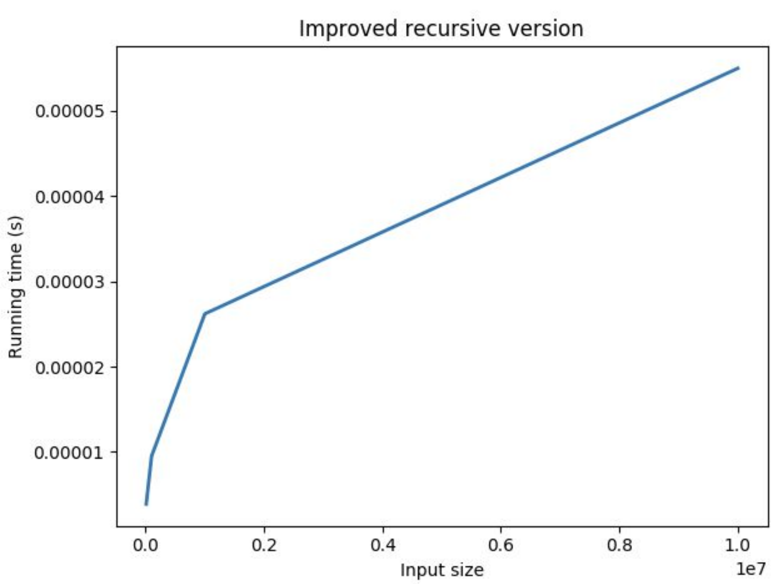
False

Else:

True







So, as we can see from the graphs the improved recursive algorithm is so similar with the iterative algortihm, and passes the older recursive one.